Software for the Simulation of Power Plant Processes.

Part A: The Mathematical Model. + Part B.

Elmegaard, Brian; Houbak, Niels

Published in:
Proceedings of ECOS 2002

Publication date:
2002

Document Version
Early version, also known as pre-print

SOFTWARE FOR THE SIMULATION OF POWER PLANT PROCESSES. PART A.

Brian Elmegaard
Department of Mechanical Engineering
Technical University of Denmark (DTU)
Nils Koppels Alle, Building 402, DK-2800 Kgs. Lyngby, DENMARK
Phone: (+45) 4525 4169, Fax: (+45) 4593 5215, E-mail: be@mek.dtu.dk

Niels Houbak
Department of Mechanical Engineering
Technical University of Denmark (DTU)
Nils Koppels Alle, Building 402, DK-2800 Kgs. Lyngby, DENMARK
Phone: (+45) 4525 4154, Fax: (+45) 4593 5215,
E-mail: Niels.Houbak@mek.dtu.dk

ABSTRACT.
Modelling of energy systems has been increasingly more important. In particular the dynamic behaviour is critical when operating the systems closer to the limits (either of the process, the materials, the emissions or the economics, etc.). This enforces strong requirements on both the models and their numerical solution with respect to both accuracy and efficiency. In part A of this paper we give a survey on simulation of energy systems, from models and modelling, over numerical methods to implementational techniques. It covers important aspects of the different phases of modelling of a (energy) system. It also gives a short introduction to robust numerical methods which it is strongly recommended to use. Finally, a few important aspects (sparse matrix technique, handling discontinuities and table look-ups) of the implementation of an energy system simulator are described. Part B contains a short description of many static and/or dynamic energy system or process simulators. It discusses the principal implementation of the model handling in DNA and finally, a small example illustrates that too simple component models may result in an erroneous, singular model.

INTRODUCTION.
During the last ten to twenty years, there has been a tremendous development of software for modelling and simulating energy (or process) systems. The need for precise information about the behaviour of a system has increased in connection with a higher degree of integration of processes, optimization of processes, getting closer to the limits of strength of the materials due to temperature, pressure, or dimensions, or a need for better control of the processes due to for example quality of the product.

There is no ideal tool applicable for types of systems to be simulated. Different areas have different needs in terms of routines for material properties, types of problems to be solved, special output, discontinuities, etc.. The tool developer needs not know all the needs of his future costumers when the tool is designed.

In many cases these software tools will have (more or less) the same general structure. There is the (graphical) user interface - (G)UI, then there is the modelling kernel part which defines the problem that can be solved, and finally there is the numerical solution routines that actually solves the mathematical model. In this paper we will focus on the inter-play be-

tween the modelling kernel and the numerical solution part.

In our view, it is very important to separate clearly the numerical solution processes from the modelling kernel. This allows for an easy extension of the modelling capabilities of the code as long as it does not require new numerical features. Also, when new numerical codes become available, they can easily be included.

THE MODELLING PROCESS.

Simulation of a system, i.e. an energy system, is a task that normally has to be divided into several sub-tasks. Each of these tasks is equally important because the final difference between the physical reality and the simulated solution is the sum of the errors from each subtask.

The first task to be performed is to build a physical model of the real system. This model is typically a drawing of the system. The boundaries for the system are specified, and neglected processes and processes accounted for are determined. These approximations to the nature of the problem can cause a difference between the actual behaviour of the system and the computed solution.

The second task is to make a mathematical model from the physical model. This is done by applying some of the fundamental laws. Conservation of mass and energy, definition of heat exchanger effectiveness, the ideal gas equation, and the heat conduction law being examples of such fundamental laws. Some of the laws do have limitations; i.e. an ideal gas need not be ideal under the current conditions (pressure and temperature). This may influence the accuracy of the solution. At this point, it is also determined whether the model is stationary (static or time independent) or dynamic (time dependent); the choice will have a big effect on the computational effort that later will be required in order to solve the model.

The third task is to apply a numerical method to the mathematical model in order to obtain a numerical model. Finding the most appropriate numerical method depends on the type of the problem: algebraic equation system (static model), differential equation system (dynamic model), differential algebraic equation system (DAE model), or partial differential equation system (PDE model). Further, the choice of method may also depend on the actual behaviour of the problem: stiff / non-stiff, highly oscillatory or discontinuous being some important properties. It is obvious that applying the numerical method introduces errors to the solution; these are iteration error, integration error and/or discretization error.

The fourth task is to implement the numerical model in a program, to run the program, and to present the results. In this process the representation of real numbers in the computer introduces rounding errors in the solution.

The main reason for building a process simulator; i.e. a power plant simulator, is a desire to avoid going through all the four tasks every time a plant is to be modelled. Essentially, the simulator contains the mathematical models of predefined components and from these a numerical model is generated and solved. Thus, a user may actually solve rather complex problems without being an expert in numerical analysis, programming, or component models. In some simulation systems, today, the graphical user interface serves as a tool for performing task one. The danger of this automation is that the developer may not always himself fully understand the complete consequences of his decisions; thus leaving creative users with unnecessary difficulties.

It is good modelling practice to use stepwise refinement of the model. Always start with the simplest and least complicated model of the system, even if the results are known on beforehand to be too erroneous. Having verified the correct general behaviour of the simple model, new components can be added (one by one). This is useful as long as the refinements are, say, 'in one direction'. When changing direction it could be beneficial to back-track the developments to a simpler version and increase complexity in another direction from here. Later, the two directions can be com-

bined.

MATHEMATICAL MODELS OF ENERGY SYSTEMS.

In most power plant simulators, the model is often a network of inter-connected components: turbine, furnace, heat exchanger, or pump. The connections between components are assumed to be without losses. There are two methods for handling the inter-connections: either direct component-component connection or component-node-component connection. The first one has as natural consequence that the conservation laws for mass, energy and momentum are applied on each component; in the latter case nodal versions must be applied as well. In the first case, the splitting and joining of streams require separate components - in the last case joining streams is handled automatically.

The behaviour of a component is normally given by equations (constitutive relations) and preferably they should be given in residual form. Complex component models may include several table lookups in order to get correct water/steam-, gas-, or other material properties. These lookups can be expensive and should be avoided whenever possible. Other components may contain chemistry; i.e. chemical reaction equations that have to be satisfied.

The component approach requires access to a large library of precompiled components and it preferably should allow that the user can define his own components. A heat exchanger is one component but it is normally necessary to have different versions depending on the media (gas, gas composition, or liquid), the type of heat transfer (convection or radiation) or changes of phase (condensation or evaporation).

The component oriented approach usually generates many equations, but it is very user friendly. For a given component only a few parameters have to be specified, almost independent of the complexity of the underlying equations.

An alternative approach for the user, would be to generate all the equations himself. Here, the number of equations can be reduced dramatically by more or less trivial substitutions. This removes the easy parts of the system. A consequence of such substitutions might be, though, that an originally linear problem ends up being non-linear.

For most of these component based applications (in particular those with many components), it is a general observation, that only a few variables appear in any given equation. In other words, the incidence matrix with one row per equation and one column per variable is sparse (only few non-zero elements - of the order 2 to 10 per row). This can (or should) be utilized.

Many energy system models are steady state (static) models. Mathematically they can be described in the following way

(1) $\underline{0} = \underline{g}(\underline{z}, \underline{p})$.

The vector $\underline{z}$ contains all the unknown variables, $\underline{p}$ represents the parameters and the function $\underline{g}$ describes all the residual equations from the components and the conservation laws.

Sometimes, one or more of the unknown variables are determined by a differential equation. These variables are called $\underline{y}$, their time derivative $\underline{y}'$, and mathematically the system can now be written as

(2) $\underline{y}' = \underline{f}(t, \underline{y}, \underline{z}, \underline{p})$, $0 \le t \le T$, $\underline{y}(0) = \underline{y}_0$
$\underline{0} = \underline{g}(t, \underline{y}, \underline{z}, \underline{p})$, $\underline{g}(0, \underline{y}_0, \underline{z}(0), \underline{p}) = \underline{0}$

t is the model time. These equations must be solved simultaneously for $\underline{y}$ and $\underline{z}$. Systems of type (2) are referred to as DAEs in semi explicit form.

NUMERICAL METHODS.

Having a mathematical formulation of the system as in equation (1), a robust and fast method for the numerical solution is the Newton iteration. This well-known method can be described as follows

(3) $\underline{\underline{J}}\, \Delta\underline{z}_s = - \underline{g}(\underline{z}_s, \underline{p})$ $\qquad \underline{\underline{J}} = \frac{\partial}{\partial \underline{z}} \underline{g}(\underline{z}, \underline{p})$
$\underline{z}_{s+1} = \underline{z}_s + \Delta\underline{z}_s$

Subscript s is the iteration count. $\underline{\underline{J}}$ is the Jacobian matrix. The first equation in (3) is a linear system of equations that has to be solved in each iteration. It is assumed that a good initial approximation $\underline{z}_0$ to the solution exists. Specially for large systems an approximation to the exact Jacobian matrix is used because the cost of an iteration is dominated by generating the matrix and solving the linear system. There are (at least) 3 different ways of approximating the Jacobian;

1) Numerical differentiation (difference approximation),
2) rank one updates (Broyden update), and
3) assume the Jacobian matrix to be constant for several iterations.

Numerical differentiation of a scalar function in one variable can be written as follows

$$(4)\quad \frac{\partial}{\partial z} g(z, p) \simeq (g(z+\Delta, p) - g(z, p)) / \Delta \simeq (g(z+\Delta, p) - g(z-\Delta, p)) / 2\Delta$$

where the last form is a second order central difference approximation. The computational cost of generating either of these approximations are N+1 or 2N calls to the g function, respectively

For very large problems the non-linearities are often located in relatively few equations; combining 1) and 3) may be advantageous. The iteration is converging as long as the residual ($\underline{g}(\underline{y}, \underline{p})$) is decaying in each component. As long as this decay is satisfactory, there is no need to update the iteration matrix.

In many simulators, a simpler iteration technique is applied. Substituting the Jacobian matrix with the identity matrix (I), the method is called functional iteration or simultaneous iteration. In the sequential version, the solution vector is updated one component at a time (or one model component at a time) and in the parallel version, the whole solution vector is updated by $\Delta\underline{z}$ in one operation.

The ODE part of (2) is normally written as

$$(5)\quad \underline{y}' = \underline{f}(t, \underline{y}) \ , \quad 0 \le t \le T \ , \quad \underline{y}(0) = \underline{y}_0$$

and is referred to as an initial value problem. The numerical solution is only computed at special points called step points. From the initial solution the solutions at the step points are generated sequentially. By interpolation the analytical solution can be approximated between step points. The distance between two step points is called the step size (denoted h). A good, simple, and popular ODE method is the classical 4'th order Runge-Kutta.

$$(6)\quad \begin{aligned} \underline{K}_1 &= \underline{f}(t_n \ , \ \underline{y}_n) \\ \underline{K}_2 &= \underline{f}(t_n + h/2 \ , \ \underline{y}_n + h/2*\underline{K}_1) \\ \underline{K}_3 &= \underline{f}(t_n + h/2 \ , \ \underline{y}_n + h/2*\underline{K}_2) \\ \underline{K}_4 &= \underline{f}(t_n + h \ \ , \ \underline{y}_n + h \ \ *\underline{K}_3) \\ \underline{y}_{n+1} &= \underline{y}_n + h/6*(\underline{K}_1 + 2*\underline{K}_2 + 2*\underline{K}_3 + \underline{K}_4) \end{aligned}$$

Subscript n is the step number. The $\underline{K}$-values are the slopes of the solution curves at various points, and the final solution update is a linear combination (an average value) of these slope values over the step. Since the method is of order four, it locally approximates all the terms up to order four of the Taylor expansion of the true solution. That is,

$$(7)\quad le_{n+1} = y_{n+1} - y_{n,x} \simeq O(h^5)$$

where $y_{n,x}$ is the value of the analytic solution at time $t_n + h$, that passes through the point (t_n, y_n). By embedding either a higher order or a lower order method into (6) it is possible to estimate the local error (le_{n+1}) in one step of the method. It is customary to use this estimate for controlling the step size in such a way that the local error is kept below a user specified limit. Steps with a too high local error estimate are normally discarded and retried with a smaller step size. Strategies for step size control can be found in [3].

Some ODEs are stiff: Initially a steep transient dramatically varies the solution; after the transient has died out, the solution is slowly varying. The step size strategy will select a small step size during the transient and increase the step size when the transient has died out. There are various definitions of stiffness. The mathematical way of describing the above is to say that after the transient has died out, the solution will no longer contain components in the direction of the eigenvector associated to the largest (negative real part) eigenvalue. Thus, problems with a large ratio between the

largest and the smallest eigenvalue have the potential for being stiff problems. The tricky part is though, that the interval of integration [0,T] also affects whether a problem is stiff or not.

The previous Runge-Kutta method is explicit and is not well suited for solving stiff problems due to its stability properties. The step size control will restrict the step size as if the transient still exists after it has died out. By introducing implicitness in the method this can be changed. A simple implicit method is the backward Euler method

(8) $\underline{y}_{n+1} = \underline{y}_n + h\, \underline{f}(t_{n+1}, \underline{y}_{n+1})$.

The method is implicit in $\underline{y}_{n+1}$. It is rewritten in (9), which is of the same form as (1). It is solved using the quasi-Newton method previously described

(9) $\underline{0} = \underline{F}(\underline{y}_{n+1}) = \underline{y}_n + h\, \underline{f}(t_{n+1}, \underline{y}_{n+1}) - \underline{y}_{n+1}$.

This method is much more expensive per step than an explicit method since it involves both an iteration in the Newton method and the forming and solution of a linear equation system. Another way of defining stiffness could be, that a problem is stiff if it is solved faster on a given computer with an implicit method than with an explicit method.

One type of implicit ODE methods are the BDF methods. For constant step size they can be derived from (8) by adding a linear combination of backward information

(10) $c_k\underline{y}_{n-k} + c_{k-1}\underline{y}_{n-k+1} + \ldots + c_0\underline{y}_n + \underline{y}_{n+1} = b\, h\, \underline{f}(t_{n+1}, \underline{y}_{n+1})$.

In each step, the system of non-linear equations to be solved is in principle identical to (9). b and c_0 to c_k are determined in order to achieve the appropriate order (k+1). These methods are normally implemented using the Nordsieck formulation (a Taylor expansion, see [5]), but this does not change the form of the system of non-linear equations to be solved in each step. Coefficients for several of these methods can be found in [2].

Also implicit Runge-Kutta methods exists. A 2 stage, 2'nd order Diagonally Implicit Runge-Kutta (DIRK) is given in (11). The coefficients can be found in [4].

(11) $\underline{K}_1 = \underline{f}(t_n + b_1 h, \underline{y}_n + h\,\gamma\,\underline{K}_1)$
$\underline{K}_2 = \underline{f}(t_n + b_2 h, \underline{y}_n + h\, a_{21}\,\underline{K}_1 + h\,\gamma\,\underline{K}_2)$
$\underline{y}_{n+1} = \underline{y}_n + h\,(c_1\,\underline{K}_1 + c_2\,\underline{K}_2)$

with

$\gamma = 1 - 1/\sqrt{2}$ $\quad a_{21} = 9 - 28*\gamma$
$b_1 = \gamma$ $\quad b_2 = a_{21} + \gamma$
$c_1 = (43 + 10*\gamma) / 62$ $\quad c_2 = 1 - c_1$

By substituting

$\underline{Y} = \underline{y}_n + h\,\gamma\,\underline{K}_1$ (or $\underline{K}_1 = (\underline{Y} - \underline{y}_n) / (h\,\gamma)$)

into the first stage of (11) we get an equation of the form (9) with $\underline{Y}$ being the iteration variable.

Both of the above implicit methods can be embedded with higher order methods in order to obtain local error estimates for step size control.

The DAE problem (2) can be solved in one of two ways. If an implicit ODE method is used, equation (9) from the ODE part is coupled with the algebraic part of the system. For explicit methods it is possible to apply a quasi-Newton method for solving the algebraic part, each time the derivatives have to be calculated.

A special property, called the index, is associated to DAE problems. By differentiating all the equations in the algebraic part of (2) with respect to time, the system may reduce to a (simple) ODE system of the form (5), otherwise, one can make another differentiation. The index is defined as the number of times it is necessary to perform this differentiation in order to end up with a system of the form (5). The above methods for solving DAEs only works for index one problems. It turns out for higher index problems, that the algebraic part of the system matrix (∂ g(t, y, z, p) / ∂z) is singular.
DAE problems are covered in detail in [1].

IMPLEMENTATION DETAILS.

In the design of simulation software it is very essential to have a structured approach; that is, make sure that the different parts of the solution process are kept clearly separated and anyway very well integrated. This is the

contradiction between easy maintenance and efficiency, maybe. In many cases, simulation software has a remarkable long life time; thus, the increase in computational power (doubling every 18 month) makes it essential to emphasise maintainability. This is not an argument for disregarding efficiency!

A static and/or dynamic energy system simulator is a complicated piece of program with many possibilities for overlooking a problem during the design phase. Restoring such a problem afterwards can be a time-consuming job. In this section we indicate a few of the problems, that experience shows are easily overlooked.

In the design stage it is very important to try to imagine the size of the largest system ever to be solved by the simulator. Why? For large systems, the CPU time is crucial. Thus, the overhead associated with measures used in order to reduce the CPU time for large systems will often be bearable for small systems even if it doubles the CPU time. An example on such a measure is sparse matrix technique.

The solution of systems of linear equations is a kernel activity in all simulators. The storage required for all elements in a matrix of dimension N, is N^2. The number of operations required for solving a system of linear equations with the matrix increases with N^3. Thus, if most of the elements, more than 98%, say, of the matrix are zero by definition, there is much to gain by avoiding these elements. In the below Fig. 1 a possible storage scheme that holds the non-zero entries of the matrix in any order is illustrated.

A:	a_{23}	a_{14}	a_{27}	a_{33}	a_{28}	..	$a_{22,7}$
row:	2	1	2	3	2	..	22
col:	3	4	7	3	8	..	7

Figure 1: Sparse matrix storage scheme.

The storage requirements are three arrays (one real array for the values and two integer arrays for row-/column- numbers, respectively) long enough to accommodate room for all non-zero entries. This is a typical storage scheme used as user interface by standard sparse matrix software, see [6]. The scheme can be utilized almost directly by iterative linear equation solution methods. For the sake of efficiency of direct solution methods, the information is re-arranged internally such that all elements in one row are stored in consecutive locations.

It should be mentioned, that in a sparse matrix framework, the generation of the sparse Jacobian matrix by divided differences (4) can benefit from having access to the structure of the matrix on before-hand. The structure, or a major approximation to it, could be available from an initial analysis of the system to be simulated. It is clear that if two variables do not appear in the same equations they can be perturbed simultaneously. Analytic evaluation of the Jacobian matrix is an alternative, but when generating a new component model, all derivatives including derivatives of state variables from, say, steam tables, must be generated as well.

The effect of using sparse matrix software is often a CPU time that increases only slightly more than linearly with the number of components in a system model. For small systems, the CPU time spent on linear algebra is insignificant, thus a major overhead resulting from the sparse matrix code is normally acceptable.

The linear algebra is used for the iterative solution of non-linear equations, normally. Since a non-linear model may have 0, 1, or several solutions, this part of the code must be designed with care. For the sake of reliability, it is important that the simulator allows for a continuous validation of the solution against user-given bounds on variables and against internal component-given restrictions. When such a validation fails, the code must decide whether the simulation must be stopped with an appropriate error message or if some standard fix-up could be taken and the calculations continued.

In particular for dynamic simulations it is very important to reduce the number of calls to routines that calculate material properties, i.e. steam tables for power plant simulations. Almost any effort to avoid re-calculations of previous values pays off. For example, transfer values between two connected components instead of calculating the same values at each end of the connection. Also, if non-linearities are concentrated to a few components, then after a few iterations many component variables have obtained their final values. In this and similar cases, there is no need to re-calculate already known values. In some cases, accuracy of tabular values is crucial and this costs CPU time; but when accuracy is not so essential it is important to be able to switch to tabular functions that are simpler (based on the ideal gas equation) and cheaper.

A problem, that is closely related to dynamic simulations, is that of dealing with discontinuities in a model. An on-off control, say, can cause serious troubles for the step size strategy in the ODE solver, as can a linearly interpolated table of data. A discontinuity can only be passed by the integrator with a very small step size. The process of both reducing the step size and locating the point of discontinuity will cost many rejected steps. The technique to be used for reducing the cost for handling discontinuities is as follows:

1. Implement all component models such that the discontinuity is not activated during normal stepping of the integrator.
2. Before a step is accepted, all discontinuities must be checked to see if a passage has occurred.
3. When this happens, locate the point where it happened by interpolation and restart the integration from there.

Easy location of the discontinuity can be implemented using functions that change sign when a discontinuity appears. Locating a discontinuity is then a question of locating a zero of a function within the interval defined by the two end-points of the step currently being computed. Even though this can dramatically reduce the cost of passing a discontinuity further reduction may be obtained by dividing discontinuities into two categories - time given discontinuities and discontinuities that depends on the solution. The first type immediately allows the point of discontinuity to be hit precisely and only the last type needs iteration to locate the point.

Most of the above have been considered during the design of the DNA code. Further details are given in part B. In [7-8] there is a much more detailed description as well as a users guide.

CONCLUSION.

When designing another energy system simulator from the very beginning it is very important to perform a relatively thorough basic problem analysis in order to have a well designed basis for constructing the simulator. Spending time on using existing simulators in order to find advantages and disadvantages in relation to the problems that are to be solved, is very important. Experience shows, that being forced to make dramatic design changes in order to incorporate a certain feature, is very expensive. A code designed for static problems may only with severe difficulties be changed to also handle dynamic problems.

Try to be very optimistic with respect to the size (number of variables, components, equations, etc.) that the simulator should be able to solve within a reasonable time. At some point in time, there will be a user asking for even more!

Try to use as much standard software as possible - for several reasons: 1) there are absolutely no reasons for inventing the wheel again, 2) routines for solving standard problems can be obtained from independent sources and are anyway almost directly interchangeable, and 3) it requires a special expertise to develop more effective software even for special problems than standard software available from various libraries. Only in cases where standard software does not exist, consider building your own. In this situation, still consider the problem to be solved using 'standard tasks'. The resulting modularisation will be invaluable in the future maintenance and extension of the code.

When modelling large and complex systems it is necessary to have the right tools. This can be either a set of basic numerical solution routines or a dedicated simulator. Even with the right tools, it is still necessary to be very careful when building and validating a model.

Equally important is the step by step refinement of a model. Develop and test sub-models before they are glued together to form a complete system model. Modern simulation tools have many facilities that are very useful for making good models and validating them, but these features can never take the responsibility for the quality of a model away from the modeler.

REFERENCES.

[1] Brenan, K.E., Champbell, S.L., and Petzold, L.R.: "Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations". North-Holland. 1989.

[2] Gear C.W.: "Numerical Initial Value Problems in Ordinary Differential Equations". Prentice-Hall. 1971.

[3] Gustafsson K.: "Control of Error and Convergence in ODE Solvers". Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden. 1992. (Ph.D thesis)

[4] Nørsett, S.P.: "Semi explicit Runge-Kutta Methods". Mathematics and Computation. Vol 6/74, NTH, ISBN 82-7151-009-6. 1974.

[5] Curtis,A.R.: Numerical Methods for Solving Stiff Systems of Ordinary Differential Equations - or - is there a life after Gear? NPL report DITC 29/83, UK, 1983.

[6] Duff, I.S.: Sparse Matrices and their Uses. Academic Press, 1981.

[7] Elmegaard, B.: Simulation of Boiler Dynamics - Development, Evaluation and Application of a General Energy System Simulation Tool. PhD thesis, Technical University of Denmark, 1999.

[8] Lorentzen, B.: Power Plant Simulation. PhD thesis, Technical University of Denmark, Laboratory for Energetics, 1995.

NOMENCLATURE.

A: Array holding matrix entries.
$\underline{F}(\ )$: A (vector) function.
$\underline{J}$: Jacobian matrix.
$\underline{\underline{K}}_i$: Stage i values in Runge-Kutta methods.
N: Number of equations.
T: Final integration time.
$\underline{Y}$: Iteration variable (local approximation to solution.

a: Matrix element or R-K method coefficient.
b, c: ODE method coefficients.
col: Array holding column numbers.
$\underline{f}(\ )$: A (vector) function of some variables.
$\underline{g}(\ )$: A (vector) function of some variables.
h: Step size in ODE method.
$\underline{le}$: Local error estimate.
$\underline{p}$: Vector of parameter values.
row: Array holding row numbers.
t: Integration time.
$\underline{y}$: Dynamic variables.
$\underline{z}$: Static variables.

Δ: Increment in the following vector
γ: Particular method constant.

Subscripts.
n: Integration step number.
s: Iteration counter.
x: Exact solution.
0: Initial value.